

Design & Analysis of Algorithms

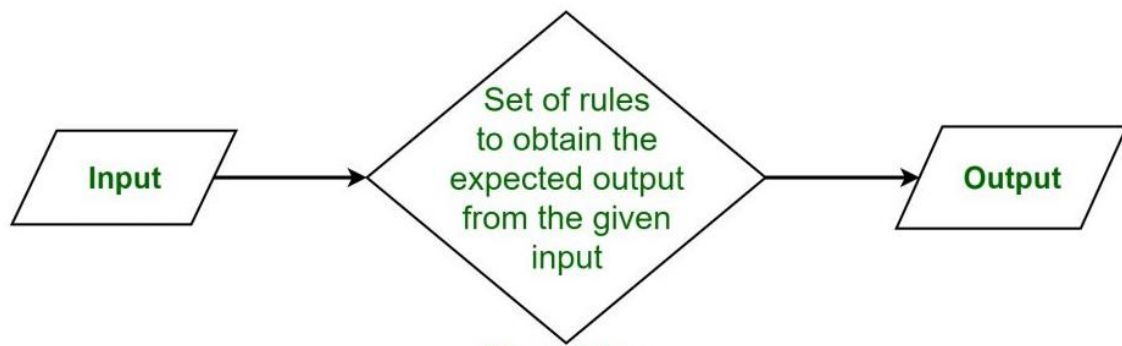
Introduction to Algorithms

Definition of Algorithm

The word Algorithm means " A set of finite rules or instructions to be followed in calculations or other problem-solving operations "

Or

" A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations". Therefore Algorithm refers to a sequence of finite steps to solve a particular problem.



Use of the Algorithms:

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

1. **Computer Science:** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
2. **Mathematics:** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.
3. **Operations Research:** Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.

Design & Analysis of Algorithms

4. **Artificial Intelligence:** Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.
5. **Data Science:** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

What is the need for algorithms?

1. Algorithms are necessary for solving complex problems efficiently and effectively.
2. They help to automate processes and make them more reliable, faster, and easier to perform.
3. Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.

Types of Algorithms

Understanding the different types of algorithms can help in selecting the most appropriate one for solving a specific problem. Broadly algorithms are categorized on the basis of their use cases and their structural or problem-solving strategies:

Algorithm Use Cases

- **Search algorithms.** Designed to retrieve information stored within some data structure, e.g., binary search algorithm used to find a particular item in a sorted list.
- **Sorting algorithms.** They rearrange the elements of a dataset in a specified order, like quicksort and merge sort, which are efficient for sorting large datasets.

Design & Analysis of Algorithms

- **Graph algorithms.** These deal with graphs, which are mathematical structures used to represent pairwise relations between objects, e.g. **Dijkstra's algorithm** finds the shortest path between nodes in a graph.

Structural or Problem-solving Strategies

- **Dynamic programming algorithms.** Implemented to solve problems by breaking them down into smaller sub problems.
- **Brute force algorithms.** By trying all possible solutions until the correct one is found, brute force algorithms can be effective, but time-consuming for complex problems.
- **Recursive algorithms.** These algorithms call themselves with smaller input values and use the results of these calls to solve the current problem
- **Greedy Algorithms.** Greedy algorithms make locally optimal choices at each step with the hope of finding the global optimum. One example is the Huffman coding algorithm, used for lossless data compression.
- **Divide and conquer algorithms.** These algorithms divide the problem into smaller sub problems, solve them independently, and then combine their solutions to solve the original problem. The merge sort algorithm is a classic example of a divide and conquers strategy.

What Makes a Good Algorithm?

There are several principles that underpin whether an algorithm is effective and fit for use:

Correctness. Foremost, a good algorithm must be correct, meaning it should always produce the right output for any given input. It should be free of errors and bugs to ensure reliable performance.

Efficiency. Efficiency is a critical aspect of a good algorithm. It refers to the optimal use of computational resources, including time and memory

Design & Analysis of Algorithms

Simplicity. A good algorithm should be simple and straightforward, avoiding unnecessary complexity.

Flexibility. Flexibility is the ability of an algorithm to adapt to changes and varying conditions. A flexible algorithm can accommodate different inputs and adjust to modifications without compromising its performance.

Robustness. Robustness refers to the algorithm's ability to handle errors gracefully. A robust algorithm can manage unexpected inputs or conditions without crashing, providing stable and reliable performance.

Stability. Stability is crucial; it ensures that the algorithm performs reliably and consistently under various conditions, maintaining its accuracy and reliability over time, even with varied inputs.

Maintainability. Maintainability is about how easily an algorithm can be updated or modified. A maintainable algorithm allows for smooth updates and an alteration, ensuring it remains up-to-date and functional over time.

Documentation. Good algorithms come with comprehensive documentation that outlines how the algorithm works, its limitations, and how to use it effectively..

Security. A good algorithm should be designed with security in mind, ensuring that it protects sensitive data and resists attacks from malicious entities.

Design & Analysis of Algorithms

Asymptotic Notation

Asymptotic notations in data structure are mathematical tools used to analyze the runtime of algorithms as input size grows. They enable comparison of algorithm efficiencies without manual runtime calculations, particularly beneficial for larger inputs. Asymptotic notations offer crucial insights into algorithm performance, aiding in algorithm selection and optimization.

There are mainly three asymptotic notations:

1. **Big-O Notation (O-notation)**
2. **Omega Notation (Ω -notation)**
3. **Theta Notation (Θ -notation)**

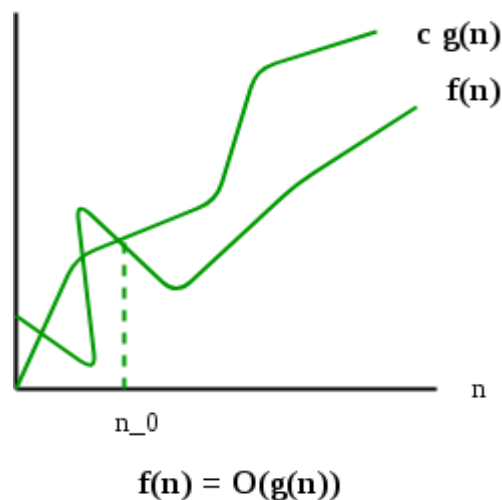
Big-O Notation:

If $f(n)$ and $g(n)$ be two non-negative functions and there exists n_0 and a constant $c > 0$ such that for all integers $n > n_0$.

$$f(n) \leq c \cdot g(n)$$

then $f(n)$ is Big Oh of $g(n)$. this is denoted as $f(n) \in O(g(n))$

it express the upper bound of algorithm running time .



Design & Analysis of Algorithms

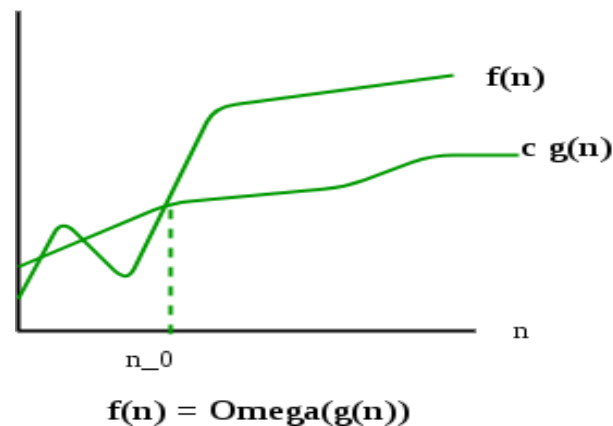
Big Omega Notation (Ω -notation)

If $f(n)$ and $g(n)$ be two non-negative functions and there exists n_0 and a constant $c > 0$ such that for all integers $n > n_0$.

$$f(n) \geq c \cdot g(n)$$

This is denoted as $f(n) \in \Omega(g(n))$

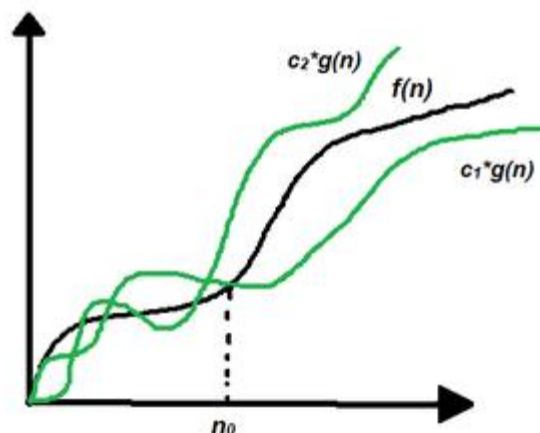
It express the lower bound of algorithm running time .



Theta Notation (Θ -notation)

If $f(n)$ and $g(n)$ be two non-negative functions and there exists n_0 and a positive constant c_1 and c_2 ($c_1 > 0$ & $c_2 > 0$) such that for all integers $n > n_0$.

Then $c_1 g(n) \leq f(n) \leq c_2 g(n)$ then $f(n) \in \Theta(g(n))$.



Design & Analysis of Algorithms

Questions about Asymptotic notations:

Find the O notation for $f(n)=10n^2+7$

Find the Ω notation for $f(n)=27n^2+16n+25$

Find the Θ notation for $f(n)=27n^2+16$

Conditions for Asymptotic Notation:

O notation:

$$f(n) \leq c.g(n)$$

Ω notation:

$$f(n) \geq c.g(n)$$

Θ notation:

$$c_1g(n) \leq f(n) \leq c_2g(n)$$

Properties of Asymptotic Notation

Reflexive

Symmetric

Transitive

Design and Analysis of Algorithm

Binary Search Tree

Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner. Binary search tree follows all properties of binary tree and its left child contains values less than the parent node and the right child contains values greater than the parent node. This hierarchical structure allows for efficient Searching, Insertion, and Deletion operations on the data stored in the tree. **Binary Search Tree (BST)** is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

The Anatomy of Binary Search Trees

The different components of a binary search tree in the data structure are as follows:

- **Nodes:** Node refers to a termination point in any binary search tree in data structures.
- **Roots:** The node on top of a binary tree is called the root.
- **Leaf Node:** The external nodes with no child are called the leaf nodes.
- **Internal Node:** All inner nodes with at least one child node are called internal nodes.
- **Parent:** Apart from the root, all other nodes of the binary tree with a minimum of one subnode are a parent.
- **Child:** The node rising straight from the parent node is the child.
- **Edge:** An edge can indicate a relationship between two nodes by connecting them.
- **Depth/ Height of a Tree:** The root or tree height signifies the highest number of edges from the farthest leaf node to the edges.

Design and Analysis of Algorithm

The binary search tree data structure can support different basic operations like:

- Search
- Insert
- Find minimum
- Find maximum
- Delete
- Find the kth smallest element
- Find successor
- Find predecessor

Implementation of Binary Search Tree

The binary search tree in data structures gets implemented in the following steps:

Step 1: Read the user's search element.

Step 2: Look for the middle element in the sorted list.

Step 3: Use the middle element in the sorted list to compare the search element.

Step 4: When both get matched, it shows "Given element is found!!!" The function also gets terminated.

Step 5: When the elements don't match, it's necessary to determine whether the search element is larger or smaller than the middle element.

Step 6: When the search element is smaller than the middle element, it becomes necessary to execute steps 2,3,4 and 5 for the middle element's left sublist.

Step 7: When the search element is larger than the middle element, it becomes necessary to repeat steps 2, 3, 4, and 5 for the middle element's right sublist.

Design and Analysis of Algorithm

Step 8: If that element also fails to match the search element, it shows “Element is not found in the list!!!” After that, the function gets terminated.

Types of Binary Search Trees

The different types of binary search trees are as follows:

- **Balanced binary tree:** In a balanced binary search tree, the height of the right and left subtrees of a node don't differ by more than 1.
- **Full binary tree:** Each node in a full binary tree has two or zero children.
- **Complete binary tree:** In a complete binary search tree, all tree levels, except for the lowest one, are occupied by nodes.
- **Perfect binary tree:** The internal nodes of a perfect binary tree have a maximum of two children. Moreover, every leaf node remains at the same level within the tree.
- **Degenerate binary tree:** In a degenerate or pathological binary tree, every node has a single child.

Example:

Create binary search tree with following keys:

4,2,3,6,5,7,1

Design and Analysis of Algorithm

Disjoint Set (Union-Find Algorithm)

Two sets are called **disjoint sets** if they don't have any element in common, the intersection of sets is a null set.

A data structure that stores non overlapping or disjoint subset of elements is called disjoint set data structure. The disjoint set data structure supports following operations:

- Adding new sets to the disjoint set.
- Merging disjoint sets to a single disjoint set using **Union** operation.
- Finding representative of a disjoint set using **Find** operation.
- Check if two sets are disjoint or not.

Operations on Disjoint Set Data Structures:

1. Find
2. Union

1. Find:

Can be implemented by recursively traversing the parent array until we hit a node that is the parent of itself.

Time complexity: This approach is inefficient and can take $O(n)$ time in worst case.

2. Union:

It takes **two elements** as input and finds the representatives of their sets using the **Find** operation, and finally puts either one of the trees (representing the set) under the root node of the other tree.

Time complexity: This approach is inefficient and could lead to tree of length $O(n)$ in worst case.

Design and Analysis of Algorithm

String Matching Introduction

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm is declared as "this is the method to find a place where one or several strings are found within the larger string."

Applications of String Matching Algorithms:

Plagiarism Detection: The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.

Bioinformatics and DNA Sequencing: Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.

Algorithms of string Matching

- **Naive Algorithm:** It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
- **KMP (Knuth Morris Pratt) Algorithm:** The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.
- **Rabin Karp Algorithm:** It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

NP Complete Problem

NP Problem:

Design and Analysis of Algorithm

The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time.

NP-Hard Problem:

A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.

If every problem of NP can be polynomial time reduced to it called as NP Hard.

A lot of times takes the particular problem solve and reducing different problems.

example :

1. Hamiltonian cycle .
2. optimization problem .
3. Shortest path

NP-Complete Problem:

A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.

A problem is np-complete when it is both np and np hard combines together.

this means np complete problems can be verified in polynomial time.

Design and Analysis of Algorithm

Difference between NP-Hard and NP-Complete:

NP-hard	NP-Complete
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, it do not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Time is unknown in NP-Hard.	Time is known as it is fixed in NP-Hard.
NP-hard is not a decision problem.	NP-Complete is exclusively a decision problem.
Not all NP-hard problems are NP-complete.	All NP-complete problems are NP-hard
Do not have to be a Decision problem.	It is exclusively a Decision problem.
It is optimization problem used.	It is Decision problem used.

Approximation Algorithm

Design and Analysis of Algorithm

An Approximate Algorithm is a way of approach NP-COMPLETENESS for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

Performance Ratios

The main idea behind calculating the **performance ratio** of an approximation algorithm, which is also called as an **approximation ratio**, is to find how close the approximate solution is to the optimal solution.

The approximate ratio is represented using $\rho(n)$ where n is the input size of the algorithm, C is the near-optimal solution obtained by the algorithm, C^* is the optimal solution for the problem. The algorithm has an approximate ratio of $\rho(n)$ if and only if –

$$\max\{C/C^*, C^*/C\} \leq \rho(n)$$

Few popular examples of the approximation algorithms are –

- Vertex Cover Algorithm
- Set Cover Problem
- Travelling Salesman Problem (Approximation Approach)
- The Subset Sum Problem

Polynomials and Matrix Calculation

Polynomials are expressions that must contain at least one term. Each term of a polynomial must contain at least one constant or at least variable and can contain any number of constants or variables. In each term that has multiple constants, each value is multiplied together. Exponents on variables must be positive integers. Terms can only be divided by a constant. Terms of a polynomial will be added or subtracted together.

Design and Analysis of Algorithm

Example:

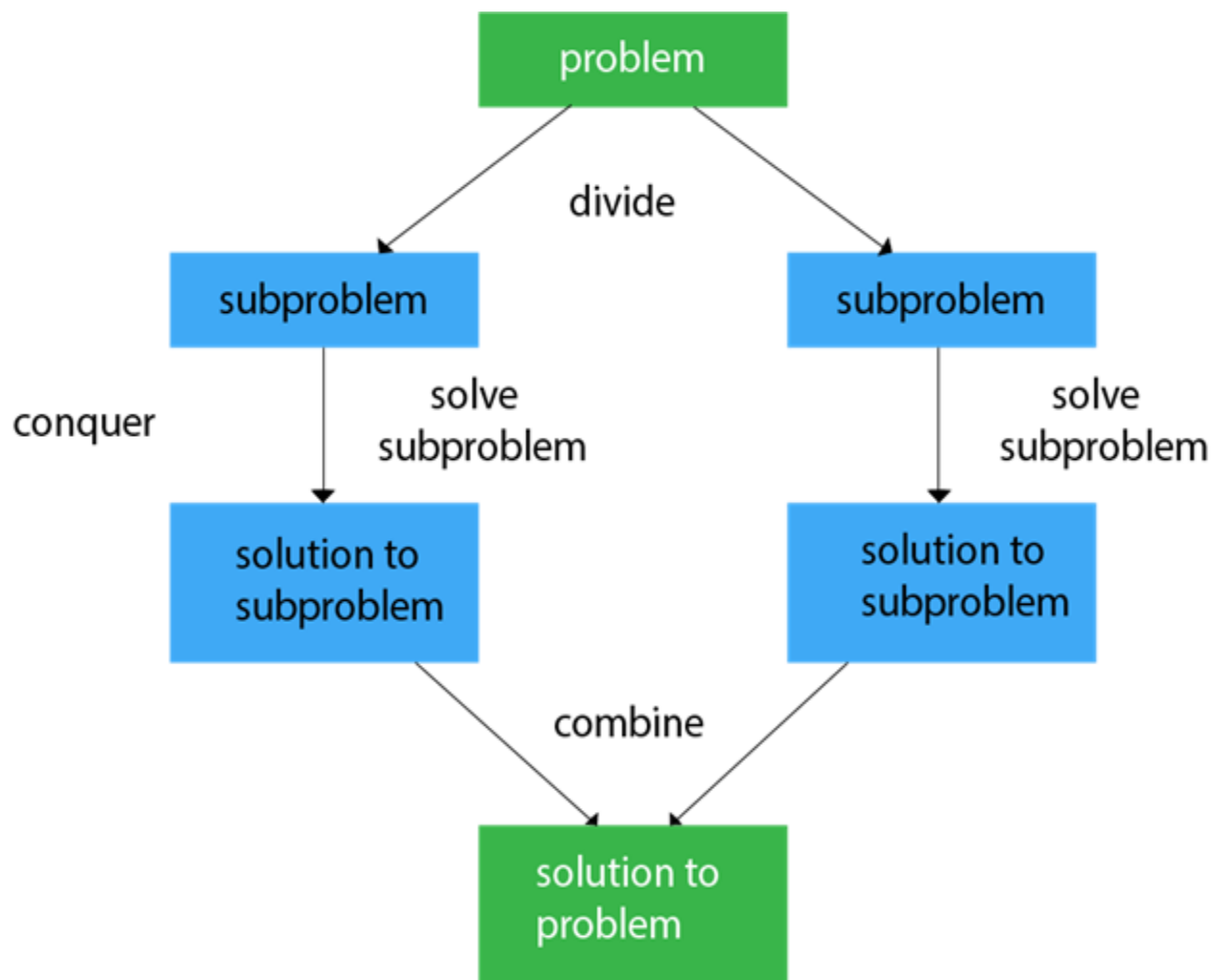
$$X^2 - 3x + 2I$$

Divide and Conquer Algorithm

Divide and Conquer Algorithm is a problem-solving technique used to solve problems by dividing the main problem into sub problems, solving them individually and then merging them to find solution to the original problem. In this article, we are going to discuss how Divide and Conquer Algorithm is helpful and how we can use it to solve problems.

Working of Divide and Conquer Algorithm:

Divide and Conquer Algorithm can be divided into three steps: **Divide**, **Conquer** and **Merge**.



1. Divide:

- Break down the original problem into smaller sub problems.
- Each sub problem should represent a part of the overall problem.
- The goal is to divide the problem until no further division is possible.

2. Conquer:

- Solve each of the smaller sub problems individually.
- If a sub problem is small enough (often referred to as the “base case”), we solve it directly without further recursion.
- The goal is to find solutions for these sub problems independently.

3. Merge:

- Combine the sub-problems to get the final solution of the whole problem.
- Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.
- The goal is to formulate a solution for the original problem by merging the results from the sub problems

Algorithm:

DAC (P)

{

If

(Small (P))

{

S(P);

}

Else

{

Divide P into P1 P2 P3PK

Apply DAC (P1), DAC (P2).....DAC (PK)

Combine DAC (P1), DAC (P2).....DAC (PK)

}

}

Application of Divide and conquer Approach

1. Binary Search
2. Quick Sort
3. Merge sort

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

- 1. Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
- 2. Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array

followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sorts the arrays recursively.

- 3. Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.

Quick Sort Example:

35 50 15 25 80 20 90 45

1. QUICKSORT (array A, start, end)
2. {
3. if (start < end)
4. {
5. p = partition(A, start, end)
6. QUICKSORT (A, start, p - 1)
7. QUICKSORT (A, p + 1, end)
8. }
9. }

Merge Sort:

Pseudo code of Merge Sort:

1. If $P < r$
2. $Q = [(p+r)/2]$
3. Merge sort (A,p,q)
4. Merge sort (A,p+1,q)

5. Merge sort (A,p,q,r)

6 4 2 1 9 8 3 5

Design and Analysis of Algorithm

Dynamic Programming

Dynamic Programming (DP) is a method used in mathematics and computer science to solve complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

How Does Dynamic Programming (DP) Work?

- **Identify Subproblems:** Divide the main problem into smaller, independent subproblems.
- **Store Solutions:** Solve each subproblem and store the solution in a table or array.
- **Build Up Solutions:** Use the stored solutions to build up the solution to the main problem.
- **Avoid Redundancy:** By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

Examples of Dynamic Programming (DP)

Example 1: Consider the problem of finding the Fibonacci sequence:

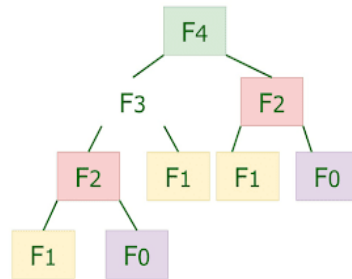
Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Brute Force Approach:

To find the n th Fibonacci number using a brute force approach, you would simply add the $(n-1)$ th and $(n-2)$ th Fibonacci numbers. This would work, but it would be inefficient for large values of n , as it would require calculating all the previous Fibonacci numbers.

Dynamic Programming Approach:

Design and Analysis of Algorithm



Fibonacci Series using Dynamic Programming

- Subproblems: $F(0)$, $F(1)$, $F(2)$, $F(3)$, ...
- Store Solutions: Create a table to store the values of $F(n)$ as they are calculated.
- Build Up Solutions: For $F(n)$, look up $F(n-1)$ and $F(n-2)$ in the table and add them.
- Avoid Redundancy: The table ensures that each subproblem (e.g., $F(2)$) is solved only once.

Approaches of Dynamic Programming (DP)

1. Top-Down Approach (Memoization):

In the top-down approach, also known as memoization, we start with the final solution and recursively break it down into smaller subproblems. To avoid redundant calculations, we store the results of solved subproblems in a memoization table.

Let's breakdown Top down approach:

- Starts with the final solution and recursively breaks it down into smaller subproblems.
- Stores the solutions to subproblems in a table to avoid redundant calculations. Suitable when the number of subproblems is large and many of them are reused.

Design and Analysis of Algorithm

2. Bottom-Up Approach (Tabulation):

In the bottom-up approach, also known as tabulation, we start with the smallest subproblems and gradually build up to the final solution. We store the results of solved subproblems in a table to avoid redundant calculations.

Let's breakdown Bottom-up approach:

- Starts with the smallest subproblems and gradually builds up to the final solution.
- Fills a table with solutions to subproblems in a bottom-up manner.
- Suitable when the number of subproblems is small and the optimal solution can be directly computed from the solutions to smaller subproblems.

Common Algorithms that Use Dynamic Programming:

- **Longest Common Subsequence (LCS):** Finds the longest common subsequence between two strings.
- **Matrix Chain Multiplication:** Optimizes the order of matrix multiplication to minimize the number of operations.
- **Fibonacci Sequence:** Calculates the nth Fibonacci number.

Advantages of Dynamic Programming (DP)

Dynamic programming has a wide range of advantages, including:

- Avoids recomputing the same subproblems multiple times, leading to significant time savings.
- Ensures that the optimal solution is found by considering all possible combinations.
- Breaks down complex problems into smaller, more manageable subproblems.

Design and Analysis of Algorithms

Greedy Approach

A greedy algorithm is a problem-solving technique that makes the best local choice at each step in the hope of finding the global optimum solution. It prioritizes immediate benefits over long-term consequences, making decisions based on the current situation without considering future implications. While this approach can be efficient and straightforward, it doesn't guarantee the best overall outcome for all problems.

Characteristics of Greedy Algorithm

Here are the characteristics of a greedy algorithm:

- Greedy algorithms are simple and easy to implement.
- They are efficient in terms of time complexity, often providing quick solutions.
- Greedy algorithms are used for optimization problems where a **locally optimal** choice leads to a **globally optimal** solution.
- These algorithms do not reconsider previous choices, as they make decisions based on current information without looking ahead.
- Greedy algorithms are suitable for problems for optimal substructure.

Examples of Greedy Algorithm

Several well-known algorithms fall under the category of greedy algorithms. Here are a few examples:

- **Dijkstra's Algorithm:** This algorithm finds the shortest path between two nodes in a graph. It works by repeatedly choosing the shortest edge available from the current node.
- **Kruskal's Algorithm:** This algorithm finds the minimum spanning tree of a graph. It works by repeatedly choosing the edge with the minimum weight that does not create a cycle.

Design and Analysis of Algorithms

- **Huffman Coding:** The greedy algorithm can be used to generate a prefix-free code for data compression, by constructing a binary tree in a way that the frequency of each character is taken into consideration.

Applications of Greedy Algorithms:

- **Dijkstra's shortest path algorithm:** Finds the shortest path between two nodes in a graph.
- **Kruskal's minimum spanning tree algorithm:** Finds the minimum spanning tree for a weighted graph.
- **Huffman coding:** Creates an optimal prefix code for a set of symbols based on their frequencies.

Dijkstra Algorithm

Dijkstra Algorithm is a graph algorithm for finding the shortest path from a source node to all other nodes in a graph (single source shortest path).

It is a type of greedy algorithm. It only works on weighted graphs with positive weights.

Working of Dijkstra's Algorithm

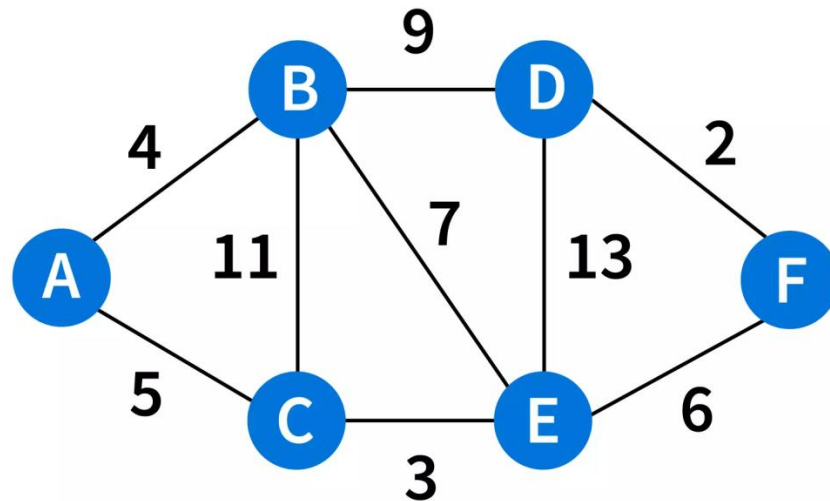
Algorithm

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node, let's say C.
3. For each neighbor N of the current node C: add the current distance of C with the weight of the edge connecting C-N. If it is smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. Go to step 2 if there are any nodes are unvisited.

Design and Analysis of Algorithms

Example:

Find the shortest path in the given weighted graph by Dijkstra algorithm.



What is the Kruskal Algorithm in minimum spanning tree?

Kruskal's algorithm is another greedy algorithm besides prim's which is used to find the minimum spanning tree from a given graph.

The main idea of Kruskal's algorithm is as follows:

Given a weighted graph with NN vertices and EE edges, the idea is to sort all the edges in increasing order of their weights, then we select the first $N-1$ edges such that they do not form a cycle within them.

It is guaranteed that those selected $N-1$ edges will construct the minimum spanning tree because there is no other way to include even smaller weight edges as we already considered the smallest weight $N-1$ edges.

Kruskal's algorithm work:

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.

Design and Analysis of Algorithms

- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

Algorithm

Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.

Step 2: Create a set E that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning

Step 4: Remove an edge from E with minimum weight

Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F

(for combining two trees into one tree).

ELSE

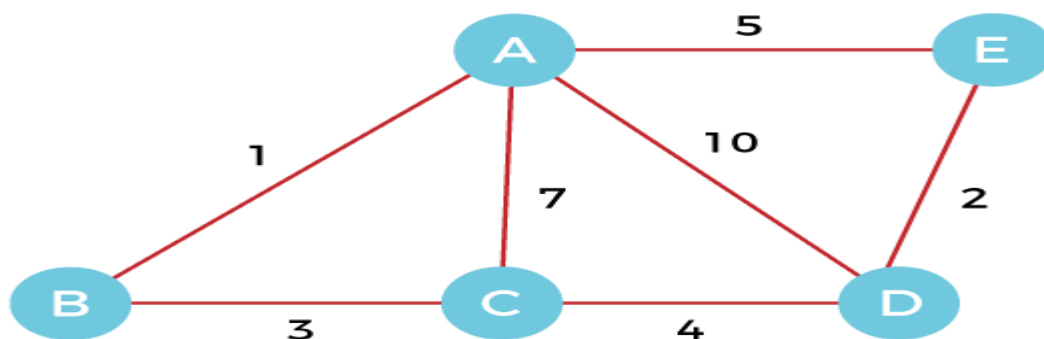
Discard the edge

Step 6: END

Kruskal Algorithm

Suppose you are given a weight graph as shown below, the task is to find the minimum spanning tree using Kruskal's algorithm, and list out all steps for finding the minimum spanning tree of the given graph.

Suppose you are given a weight graph as shown below, the task is to find the minimum spanning tree using Kruskal's algorithm, and list out all steps for finding the minimum spanning tree of the given graph.



Design and Analysis of Algorithms

Huffman Coding Algorithm

A well-known Greedy algorithm is Huffman Coding. The size of code allocated to a character relies on the frequency of the character, which is why it is referred to be a greedy algorithm. The short-length variable code is assigned to the character with the highest frequency, and vice versa for characters with lower frequencies. It employs a variable-length encoding, which means that it gives each character in the provided data stream a different variable-length code.

Prefix Rule

Essentially, this rule states that the code that is allocated to a character shall not be another code's prefix. If this rule is broken, various ambiguities may appear when decoding the Huffman tree that has been created.

Let's look at an illustration of this rule to better comprehend it: For each character, a code is provided, such as:

1. a - 0
2. b - 1
3. c - 01

Assuming that the produced bit stream is 001, the code may be expressed as follows when decoded:

1. 0 0 1 = aab
2. 0 01 = ac

What is the Huffman Coding process?

The Huffman Code is obtained for each distinct character in primarily two steps:

- Create a Huffman Tree first using only the unique characters in the data stream provided.
- Second, we must proceed through the constructed Huffman Tree, assign codes to the characters, and then use those codes to decode the provided text.

Steps to Take in Huffman Coding

Design and Analysis of Algorithms

The steps used to construct the Huffman tree using the characters provided

1. Input:
2. string str = "abbcdbccdaabbbeeebeab"

How Can a Huffman Tree Be Constructed from Input Characters?

The frequency of each character in the provided string must first be determined.

Character	Frequency
a	4
b	7
c	3
d	2
e	4

Design and Analysis of Algorithm

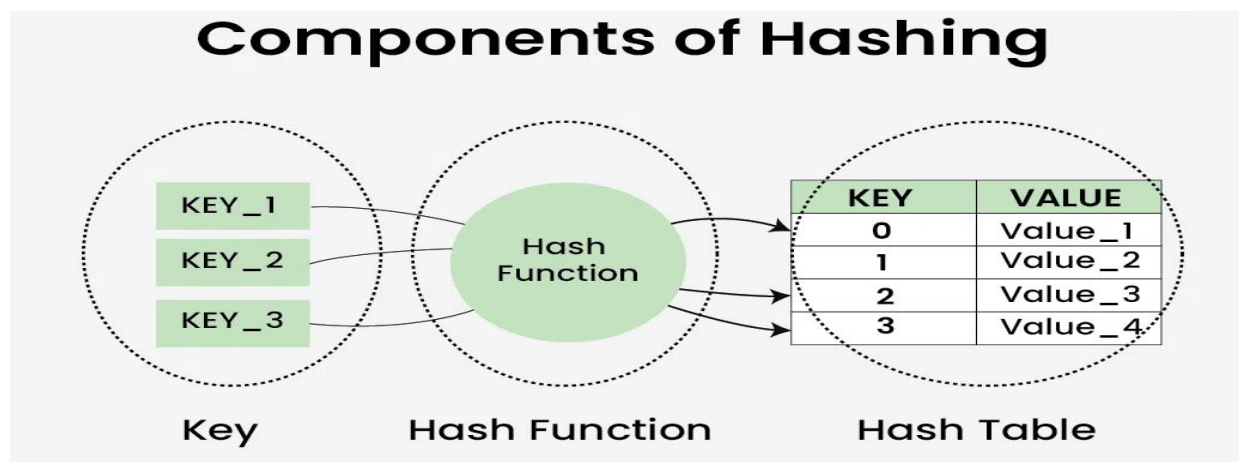
Hashing

Hashing in Data Structures refers to the process of transforming a given key to another value. It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key. The transformation of a key to the corresponding value is done using a Hash Function and the value obtained from the hash function is called Hash Code.

Components of Hashing

There are majorly three components of hashing:

1. **Key:** A **Key** can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The **hash function** receives the input key and returns the index of an element in an array called a hash table. The index is known as the **hash index**.
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function.



Algorithm:

1. Calculate the hash key. i.e. $\text{key} = \text{data} \% \text{size}$
2. Check, if `hashTable[key]` is empty

Design and Analysis of Algorithm

- store the value directly by $\text{hashTable}[\text{key}] = \text{data}$
- 3. If the hash index already has some value then
 - check for next index using $\text{key} = (\text{key} + 1) \% \text{size}$
- 4. Check, if the next index is available $\text{hashTable}[\text{key}]$ then store the value. Otherwise try for next index.
- 5. Do the above process till we find the space.

Types of Hash Functions

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. $\text{kmod}10$
2. $\text{kmod}n$
3. Mid-Square Method
4. Folding Method

Example: Let us consider a simple search key 24,52,91,67,48,83

Advantages of Hash Data structure

- Hash tables are more efficient than search trees or other data structures
- Hash provides constant time for searching, insertion, and deletion operations on average.

Disadvantages of Hash Data structure

- Hash is inefficient when there are many collisions.
- Hash collisions are practically not avoided for a large set of possible keys.

Design and Analysis of Algorithm

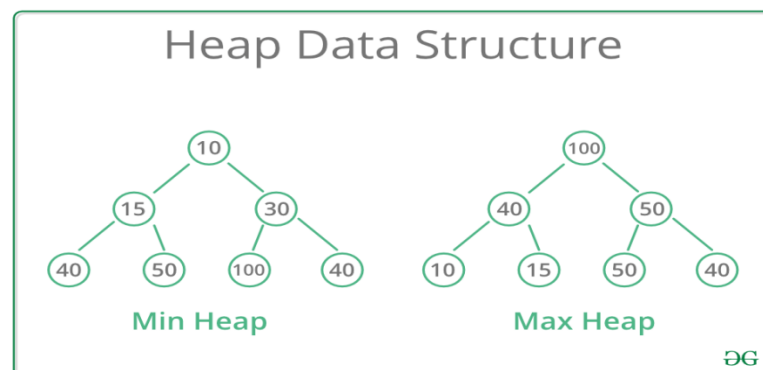
Heap Data Structure

A **Heap** is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is less than or equal to its own value. Heaps are usually used to implement priority queues, where the smallest (or largest) element is always at the root of the tree.

Types of Heaps

There are two main types of heaps:

- **Max Heap:** The root node contains the maximum value, and the values decrease as you move down the tree.
- **Min Heap:** The root node contains the minimum value, and the values increase as you move down the tree.



Heap Operations

Common heap operations are:

- **Insert:** Adds a new element to the heap while maintaining the heap property.
- **Extract Max/Min:** Removes the maximum or minimum element from the heap and returns it.
- **Heapify:** Converts an arbitrary binary tree into a heap.

Design and Analysis of Algorithm

Heap Data Structure Applications

Heaps have various applications, like:

- Heaps are commonly used to implement priority queues, where elements are retrieved based on their priority (maximum or minimum value).
- Heapsort is a sorting algorithm that uses a heap to sort an array in ascending or descending order.
- Heaps are used in graph algorithms like **Dijkstra's algorithm** and **Prim's algorithm** for finding the shortest paths and minimum spanning trees.

Network flow

Network flow is important because it can be used to express a wide variety of different kinds of problems. So, by developing good algorithms for solving network flow. A **Flow network** is a directed graph where each edge has a capacity and a flow. They are typically used to model problems involving the transport of items between locations, using a network of routes with limited capacity. Examples include modeling traffic on a network of roads, fluid in a network of pipes, and electricity in a network of circuit components.

Ford-Fulkerson Algorithm for Maximum Flow Problem

The Ford-Fulkerson algorithm is a widely used algorithm to solve the maximum flow problem in a flow network. The maximum flow problem involves determining the maximum amount of flow that can be sent from a source vertex to a sink vertex in a directed weighted graph, subject to capacity constraints on the edges.

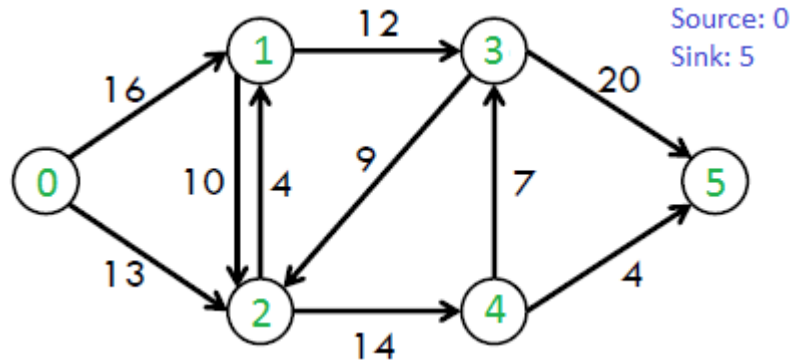
The algorithm works by iteratively finding an augmenting path, which is a path from the source to the sink in the residual graph, i.e., the graph obtained by subtracting the current flow from the capacity of each edge. The algorithm then increases the flow along this path by the maximum possible amount, which is the minimum capacity of the edges along the path.

Problem:

Given a graph which represents a flow network where every edge has a capacity. Also, given two vertices *source 's'* and *sink 't'* in the graph, find the maximum possible flow from s to t with the following constraints:

- Flow on an edge doesn't exceed the given capacity of the edge.
- Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph .



Ford-Fulkerson Algorithm

The following is simple idea of Ford-Fulkerson algorithm:

1. Start with initial flow as 0.
2. While there exists an augmenting path from the source to the sink:
 - Find an augmenting path using any path-finding algorithm, such as breadth-first search or depth-first search.
 - Determine the amount of flow that can be sent along the augmenting path, which is the minimum residual capacity along the edges of the path.
 - Increase the flow along the augmenting path by the determined amount.
3. Return the maximum flow.

Recursion and Recurrence Relation

Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. A recursive function solves a particular problem by calling a copy of itself and solving smaller sub problems of the original problems. Many more recursive calls can be generated as and when required.

Need of Recursion

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. A task that can be defined with its similar subtask, recursion is one of the best solutions for it.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

How to write recurrence relation?

Search an element using binary search algorithm for solving recurrence in sorted array .

Array:

10 20 30 40 50 60 70

i=1 j=7

x=30

Algorithm:

Bs(a,i,j,x)

Mid=(i+j)/2

If(a[mid]==x

return (mid);

else

if (a[mid]>x)

Bs(a,i,mid-1,x)

Else

Bs(a,mid+1,j,x)

Recurrence relation of this problem:

$$T(n)=T(n/2)+c$$

There are three methods for solving Recurrence:

1. Substitution Method(Back Substitution Method)
2. Recursion Tree Method
3. Master Method

The substitution method

The substitution method is based on the idea of guessing a solution for the recurrence relation and then proving it by induction. For example, suppose you have the recurrence relation $T(n) = 2T(n/2) + n$, with $T(1) = 1$. This recurrence relation describes the running time of a binary search algorithm.

The base case

The base case is the simplest case where the recurrence relation holds. Usually, this is when n is equal to the smallest value for which the recurrence relation is defined. For example, in the binary search recurrence relation, the base case is when $n = 1$. You need to show that your guessed solution matches the given value for the base case. In this case, $T(1) = 1 \log 1 = 1$, which is equal to the given value of $T(1) = 1$. Therefore, the base case is satisfied.

The induction step

The induction step is where you assume that the recurrence relation holds for some n and then show that it also holds for a larger n . Usually, this is done by substituting the recurrence relation into your guessed solution and simplifying it. For example, in the binary search recurrence relation, the induction step is to assume that $T(n) = n \log n$ for some n and then show that $T(2n) = 2n \log 2n$. To do this, you can substitute $T(2n) = 2T(n) + 2n$ into your guessed solution and simplify it as follows:

$$T(2n) = 2T(n) + 2n$$

$$= 2(n \log n) + 2n$$

$$= 2n(\log n + 1)$$

$$= 2n \log (n * 2)$$

$$= 2n \log 2n$$

Therefore, the induction step is satisfied.

Example 1:

$$T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Example 2:

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ n \cdot T(n-1) & \text{if } n > 1 \end{cases}$$

Example 3:

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \end{cases}$$

Example 4:

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1) + \log n, & \text{if } n > 1 \end{cases}$$

Recursion Tree method

The Recursion Tree Method is a way of solving recurrence relations. In this method, a recurrence relation is converted into recursive trees. Each node represents the cost incurred at various levels of recursion. To find the total cost, costs of all levels are summed up.

Steps to solve recurrence relation using recursion tree method:

1. Draw a recursive tree for given recurrence relation
2. Calculate the cost at each level and count the total no of levels in the recursion tree.
3. Count the total number of nodes in the last level and calculate the cost of the last level
4. Sum up the cost of all the levels in the recursive tree

Question 1: $T(n) = 2T(n/2) + cn$

Question 2: $T(n) = 3T(n/4) + cn^2$

Master Theorem

The Master Theorem is a tool used to solve recurrence relations that arise in the analysis of divide-and-conquer algorithms. The Master Theorem provides a systematic way of solving recurrence relations of the form:

$$T(n) = aT(n/b) + f(n)$$

1. where a , b , and $f(n)$ are positive functions and n is the size of the problem. The Master Theorem provides conditions for the solution of the recurrence to be in the form of $O(n^k)$ for some constant k , and it gives a formula for determining the value of k .
2. The advanced version of the Master Theorem provides a more general form of the theorem that can handle recurrence relations that are more complex than the basic form. The advanced version of the Master Theorem can handle recurrences with multiple terms and more complex functions.
3. It is important to note that the Master Theorem is not applicable to all recurrence relations, and it may not always provide an exact solution to a given recurrence. However, it is a useful tool for analyzing the time complexity of divide-and-conquer algorithms and provides a good starting point for solving more complex recurrences.

Master Theorem is used to determine running time of algorithms (divide and conquer algorithms) in terms of asymptotic notations.

Question no 1: $T(n) = \begin{cases} T(\sqrt{n}) + \log n & \text{if } n > 2 \\ O(1) & \text{else} \end{cases}$

Question 2: $T(n) = T(n/2) + c$

Design and analysis of algorithms

Shortest Path Algorithms

The shortest path algorithms are the ones that focus on calculating the minimum travelling cost from source node to destination node of a graph in optimal time and space complexities.

Types of Shortest Path Algorithms:

As we know there are various types of graphs (weighted, unweighted, negative, cyclic, etc.) therefore having a single algorithm that handles all of them efficiently is not possible. In order to tackle different problems, we have different shortest-path algorithms, which can be categorised into two categories:

Single Source Shortest Path Algorithms:

In this algorithm we determine the shortest path of all the nodes in the graph with respect to a single node i.e. we have only one source node in this algorithms.

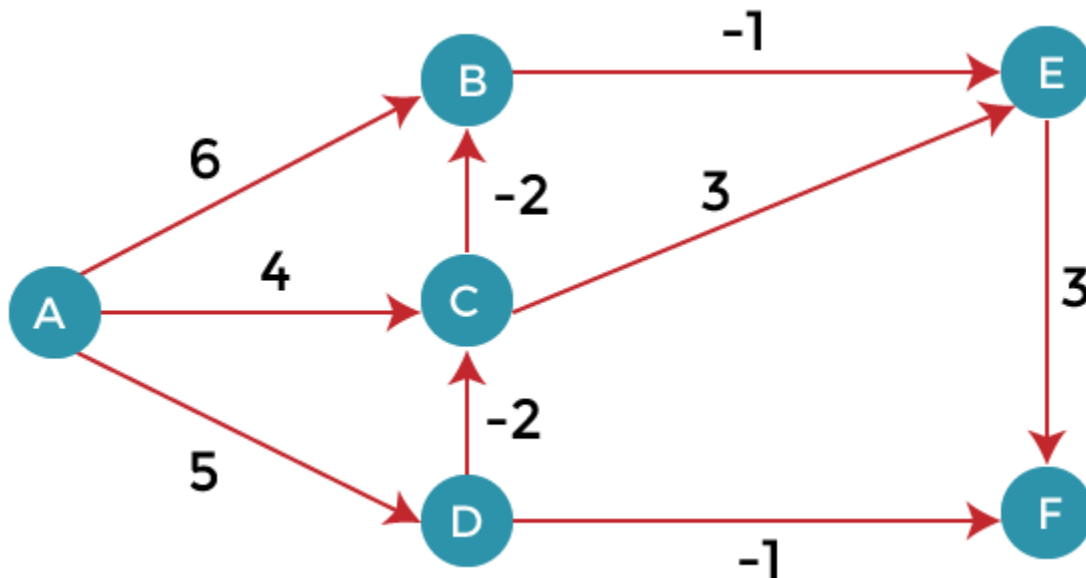
1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Dijkstra's algorithm
4. Bellman ford

Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both weighted and unweighted graphs. A **Bellman-Ford** algorithm is also guaranteed to find the shortest path in a graph, similar to **Dijkstra's algorithm**. Although Bellman-Ford is slower than **Dijkstra's algorithm**, it is capable of handling graphs with **negative edge weights**, which makes it more versatile. The shortest path cannot be found if there exists a **negative cycle** in the graph. If we continue to go around the negative cycle an

Design and analysis of algorithms

infinite number of times, then the cost of the path will continue to decrease (even though the length of the path is increasing).

Example:



All Pair Shortest Path Algorithms:

Contrary to the single source shortest path algorithm, in this algorithm we determine the shortest path between every possible pair of nodes.

- **Floyd-Warshall algorithm**

The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

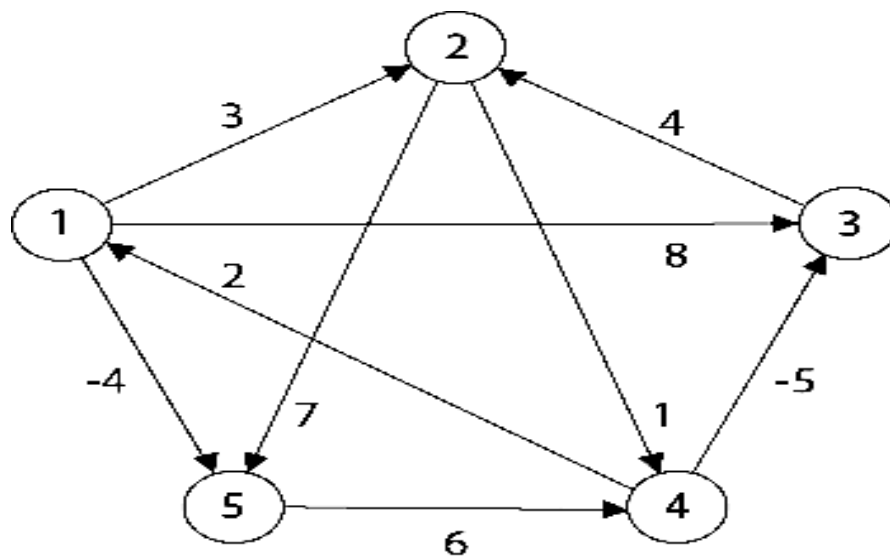
It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Design and analysis of algorithms

Floyd Warshall Algorithm Algorithm:

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices.
- For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.
 - **k** is not an intermediate vertex in shortest path from **i** to **j**. We keep the value of **dist[i][j]** as it is.
 - **k** is an intermediate vertex in shortest path from **i** to **j**. We update the value of **dist[i][j]** as **dist[i][k] + dist[k][j]**, if **dist[i][j] > dist[i][k] + dist[k][j]**

Example:



Analysis of Algorithm

Sorting Algorithms

What is Sorting?

Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. Sorting means reordering of all the elements either in ascending or in descending order.

Characteristics of Sorting Algorithms:

- **Time Complexity:** Time complexity, a measure of how long it takes to run an algorithm, is used to categorize sorting algorithms. The worst-case, average-case, and best-case performance of a sorting algorithm can be used to quantify the time complexity of the process.
- **Space Complexity:** Sorting algorithms also have space complexity, which is the amount of memory required to execute the algorithm.
- **Stability:** A sorting algorithm is said to be stable if the relative order of equal elements is preserved after sorting. This is important in certain applications where the original order of equal elements must be maintained.
- **In-Place Sorting:** An in-place sorting algorithm is one that does not require additional memory to sort the data. This is important when the available memory is limited or when the data cannot be moved.
- **Additivity:** An adaptive sorting algorithm is one that takes advantage of pre-existing order in the data to improve performance.

Applications of Sorting Algorithms:

- **Searching Algorithms:** Sorting is often a crucial step in search algorithms like binary search, Ternary Search, where the data needs to be sorted before searching for a specific element.
- **Data management:** Sorting data makes it easier to search, retrieve, and analyze.

Analysis of Algorithm

- **Database optimization:** Sorting data in databases improves query performance.
- **Machine learning:** Sorting is used to prepare data for training machine learning models.
- **Data Analysis:** Sorting helps in identifying patterns, trends, and outliers in datasets. It plays a vital role in statistical analysis, financial modeling, and other data-driven fields.
- **Operating Systems:** Sorting algorithms are used in operating systems for tasks like task scheduling, memory management, and file system organization.

Selection Sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Algorithm:

```
for (int i = 0; i < n - 1; i++)  
{  
    int min = i;  
    for (int j = i + 1; j < n; j++)  
    {  
        if (arr[j] < arr[min])  
            min = j;  
    }  
    If(min!=i)
```

Analysis of Algorithm

```
{  
swap (arr[i],arr[min]);  
}  
}
```

Example:

1)arr[] = {64, 25, 12, 22, 11}

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is $O(N^2)$ as there are two nested loops:

- One loop to select an element of Array one by one = $O(N)$
- Another loop to compare that element with every other Array element = $O(N)$
- Therefore overall complexity = $O(N) * O(N) = O(N*N) = O(N^2)$

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of $O(n^2)$ in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Analysis of Algorithm

Bubble Sort:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

In Bubble Sort algorithm,

- Traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

Bubble Sort Algorithm

```
{  
int i,j,temp;  
for ( i = 0; i < n - 1; i++)  
{  
    for (j = 0; j < n-1; j++)  
    {  
        if (arr[j] >arr[j+1]) {  
            Temp=arr[j]  
            arr[j]=arr[j+1]  
            arr[j+1]=temp  
        }  
    }  
}
```


Analysis of Algorithm

Example

`arr[] = {6, 0, 3, 5}`

Time Complexity Analysis of Bubble Sort:

Best Case Time Complexity Analysis of Bubble Sort: $O(N)$

Worst Case. Time Complexity Analysis of Bubble Sort: $O(N^2)$

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of $O(N^2)$ which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

Insertion Sort

Insertion sort is a simple sorting algorithm that works by building a sorted array one element at a time. It is considered an “**in-place**” sorting algorithm, meaning it doesn’t require any additional memory space beyond the original array.

Insertion Sort Algorithm :

{

`int i,j,num;`

Analysis of Algorithm

```
for ( i = 1; i < n ; i++)  
{  
    num=arr[j];  
    for (j = i-1; j >=0; j--)  
{  
        if (arr[j] >num)  
{  
            arr[j+1]=arr[j];  
        }  
    }  
    else  
{  
        break;  
    }  
}  
arr[j+1]=num;  
}  
}
```

Example:

Consider an array having elements: {23, 1, 10, 5, 2}

Time Complexity of Insertion Sort

- **Best case: $O(n)$** , If the list is already sorted, where n is the number of elements in the list.

Analysis of Algorithm

- **Average case: $O(n^2)$** , If the list is randomly ordered
- **Worst case: $O(n^2)$** , If the list is in reverse order

Advantages of Insertion Sort:

- Simple and easy to implement.
- Stable sorting algorithm.
- Efficient for small lists and nearly sorted lists.
- Space-efficient.

Disadvantages of Insertion Sort:

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Heap Sort :

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Heap Sort Algorithm :

{

- **build max heap**
- **for i=A length down to 2**
- {
- **Exchange $A[1]$ with $A[i]$**
- **A.heap size=A.heapsize-1**
- **Max heapify(A,1)**
- }

Analysis of Algorithm

- }

Example:

Consider the array: `arr[] = {4, 10, 3, 5, 1}`.

Complexity Analysis of Heap Sort

Time Complexity: $O(N \log N)$

Advantages of Heap Sort:

- **Efficient Time Complexity:** Heap Sort has a time complexity of $O(n \log n)$ in all cases. This makes it efficient for sorting large datasets. The **log n** factor comes from the height of the binary heap, and it ensures that the algorithm maintains good performance even with a large number of elements.
- **Memory Usage** – Memory usage can be minimal (by writing an iterative `heapify()` instead of a recursive one). So apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity** – It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

Disadvantages of Heap Sort:

- **Costly:** Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is $O(n \log n)$ for both.
- **Unstable:** Heap sort is unstable. It might rearrange the relative order.
- **Efficient:** Heap Sort is not very efficient when working with highly complex data.

Design and analysis of algorithm

Time Complexities

To evaluate and compare different algorithms, instead of looking at the actual runtime for an algorithm, it makes more sense to use something called time complexity.

Time complexity is the number of operations needed to run an algorithm on large amounts of data. And the number of operations can be considered as time because the computer uses some time for each operation.

Best, Worst, and Average Case Complexity:

In analyzing algorithms, we consider three types of time complexity:

1. **Best-case complexity ($O(\text{best})$):** This represents the minimum time required for an algorithm to complete when given the optimal input. It denotes an algorithm operating at its peak efficiency under ideal circumstances.
2. **Worst-case complexity ($O(\text{worst})$):** This denotes the maximum time an algorithm will take to finish for any given input. It represents the scenario where the algorithm encounters the most unfavorable input.
3. **Average-case complexity ($O(\text{average})$):** This estimates the typical running time of an algorithm when averaged over all possible inputs. It provides a more realistic evaluation of an algorithm's performance.

Comparison of Various types of time complexities

$O(1) < O(\log \log n) < O(\log n) < O(n^{1/2}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^k) < O(2^n) < O(n^n) < O(2^{2^n})$

Time Complexity of all searching and sorting algorithms

1. Binary search $\log_2 n$
2. Sequential search $O(n)$
3. Quick sort $O(n \log n)$

Design and analysis of algorithm

4. Merge sort $O(n \log n)$
5. Insertion sort $O(n^2)$
6. Bubble sort $O(n^2)$
7. Heap Sort $O(n \log n)$
8. Selection Sort $O(n^2)$
9. Hight of CBT $O(\log n)$
10. Insertion in heap $(\log n)$
11. Constant heap $(n \log n)$
12. Delete from heap $(\log 2n)$
13. Huffman $(n \log n)$
14. Prims $O(n^2)$
15. Kruskal $O(E \log E)$
16. DFS, BFS $O(V+E)$
17. All pair shortest $O(n^3)$
18. Dijkstra $O(V^2)$

Question 1:

$$F1(n) = n^2 \log n$$

$$F2(n) = n(\log n)^{10}$$

Question 2:

$$F1(n) = 2^n, f2(n) = n^2/3, f3(n) = n \log_2 n, f4(n) = n^{\log_2 n}$$